Stage de L3 : Terminaison en π-calcul

Pierre Boutillier
sous la direction de Sébastien Briais et Romain Demangeon
au sein de l'équipe PLUME du LIP de l'ENS Lyon

16 Juin - 28 Juillet 2008

Introduction

Lorsque qu'un programme est garant de la sécurité de personnes, il est crucial de garantir qu'il est correct, d'avoir une *preuve de correction*. Formellement, un programme est *correct* si son exécution respecte sa sémantique, c'est-à-dire vérifie une formule mathématique donnée entre ses entrées et ses sorties.

Aujourd'hui, les demandes en calcul explosent. Pour y répondre, les programmes sont écrits pour s'éxécuter sur plusieurs calculateurs travaillant en parallèle. On parle de *calcul concurrent*.

Or, le parallélisme ajoute un grand degré de complexité aux preuves de correction. En effet, la durée nécessaire pour réaliser une opération élémentaire dépend d'énormement de paramètres (nature intrinsèque de l'opération, encombrement du réseau, état de la mémoire, nature et disponibilité du processeur, ...). Deux opérations faites en parallèle ne vont donc pas finir dans un ordre déterminé. Les opérations qui leur succèdent vont donc commencer dans un ordre aléatoire : l'ordre des opérations appelé le *flot de calcul* varie donc suivant les exécutions. Ceci donne lieu à une certaine forme d'indéterminisme ; il faut alors vérifier la sémantique pour toutes les exécutions possibles.

Le problème de la correction se découpe en plusieurs sous problèmes plus spécifiques. Certaines applications doivent par exemple s'exécuter en un temps fini, il s'agit de la propriété de *terminaison*. C'est la propriété que nous allons étudier.

Dans le modèle de programmation concurrente que nous avons choisi, ce problème est indécidable. Une solution est d'imposer aux programmes des règles trop exigeantes afin d'assurer leur terminaison ; certains programmes terminants vont donc être inévitablement rejetés quelle que soit la finesse de nos règles.

Le but de ce stage était de se documenter sur les différentes propriétés pouvant assurer la terminaison puis d'implémenter un vérificateur automatique de ces propriétés. Ce vérificateur permet par exemple de vérifier que les procédés proposés pour transformer un programme séquentiel en un programme concurrent conservent la terminaison.

Dans ce rapport, la première partie définira d'abord le formalisme de calcul utilisé puis expliquera les propriétés sur lesquelles s'appuyer pour garantir la terminaison avant d'exposer les raffinements de ces principes auxquels je me suis intéressé.

La seconde partie se concentrera sur mon travail en décrivant l'interpréteur dont je suis parti et la manière dont j'ai codé le vérificateur de terminaison.

1 π-calcul et typage

Pour raisonner sur les programmes, il faut un formalisme adéquat qui modélise le comportement effectif du calcul et permette d'écrire des propriétés. Dans le cadre du calcul distribué, c'est le *π-calcul* qui est utilisé. L'explication de son fonctionnement sera l'objet du début de cette partie.

Pour garantir la terminaison, nous nous appuyons sur un *système de types*.

Le principe d'un système de type pour la propriété P est d'établir une liste de règles afin d'associer inductivement une expression appelé *type* à un programme. Si le mé-

canisme aboutit, alors le programme est dit *bien typé*. Les règles sont telles qu'un programme bien typé vérifie la propriété P.

Pour la terminaison, cette méthode a été utilisée avec succès pour le λ-calcul (modèle du calcul séquentiel) où des résultats puissant ont été obtenus, par exemple à l'aide de la réalisabilité. Le cas du calcul concurrent est l'objet des études actuelles dont celles menées par mes encadrants.

L'usage des systèmes de types ne se limite pas à garantir la terminaison. Il existe par exemple des systèmes de types qui garantissent l'équité entre clients d'un serveur ou qui assurent à l'utilisateur la cohérence de son programme et en facilite le débuggage.

1.1 Sémantique opérationnelle du π-calcul

A la base du π-calcul se trouve la notion de communication. Un programme est un ensemble d'éléments appelés *processus*. Ces éléments cherchent à recevoir ou délivrer un message et s'ils y parviennent, ils sont substitués par d'autres processus qu'on appelle leur *continuation* dans l'ensemble des processus en train de s'executer.

Les communications ont lieu sur des *canaux* représentés par un nom sur lequel il est possible d'écouter ou d'émettre un message. Ces *messages* sont des n-uplets de noms de canaux.

Si a est un nom de canal et u un message, l'envoi de u sur le canal a est noté $\bar{a}\langle u\rangle$. La réception d'un message sur un canal a nécessite de lier une certaine variable x afin de pouvoir faire référence au futur message reçu dans la continuation : elle est notée $a(x)$.

Lors d'une communication, la substitution des variables liées x de la réception par le message u de l'émission dans la continuation C de la réception est noté $C[u/x]$.

Le *processus* constitué d'une action A, écoute ou émission, suivie d'une continuation T est noté $A.T$. Les continuations et les programmes sont aussi appelés les *termes* du π-calcul. Un terme formé de l'ensemble de processus $\{P, Q, R\}$ est noté $P \mid Q \mid R$ afin de dénoter les exécutions parallèles.

Pour comprendre, prenons un programme simple : si on reçoit le message constitué du canal x sur le canal a alors on renvoie le message sur les canaux a et b et on écoute sur x pour recevoir le message m. Ce programme est noté $a(x).(\bar{a}\langle x\rangle \mid \bar{b}\langle x\rangle \mid x(m))$

L'exécution d'un terme est une succession de communications entre processus voulant émettre et recevoir sur un même canal. Lorsqu'un processus communique, il libère sa continuation. Voici un exemple d'une exécution concernant cinq processus :

$$\begin{aligned} b(y).P \mid a(x).Q \mid \bar{a}\langle u\rangle.R \mid \bar{b}\langle v\rangle.S \mid T &\longrightarrow b(y).P \mid Q[u/x] \mid R \mid \bar{b}\langle v\rangle.S \mid T \\ &\longrightarrow P[v/y] \mid Q[u/x] \mid R \mid S \mid T \end{aligned}$$

Cette exécution n'est pas unique car le terme originel aurait pu d'abord effectuer la communication sur b puis celle sur a. De plus, des processus en concurrence pour l'émission d'un message sont source de nn déterminisme. Ainsi le terme $a(x).P \mid \bar{a}\langle u\rangle.Q \mid \bar{a}\langle v\rangle.R$ peut se réduire en : $P[u/x] \mid Q \mid \bar{a}\langle v\rangle.R$ ou $P[v/x] \mid \bar{a}\langle u\rangle.Q \mid R$. Dans le dernier cas, le processus Q ne peut s'exécuter pas tant qu'il n'aura pas reçu un message sur le canal a.

Ainsi défini, tout calcul termine nécessairement car la longueur d'un terme décroît strictement avec un pas de calcul : deux actions sont consommées à chaque réduction.

Cependant, afin de modéliser un serveur effectuant le même calcul à chaque requête d'un client, on ajoute au calcul un opérateur de *réplication* noté '!' et défini par $!a(x).P$ équivaut à $!a(x).P \mid a(x).P$. Ainsi, $!a(x).P$ représente une infinité de $a(x).P$. Il n'y a pas de concurrence pour acceder à une ressource répliqué. Dans notre exemple, si on ajoute une réplication $!a(x).P \mid \bar{a}\langle u\rangle.Q \mid \bar{a}\langle v\rangle.R$ devient $!a(x).P \mid P[u/x] \mid Q \mid \bar{a}\langle v\rangle.R$ puis $!a(x).P \mid P[u/x] \mid P[v/x] \mid Q \mid R$.

Pour comprendre sur un exemple complet, prenons le terme $!a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{a} \mid \bar{b}$ (on considère ici des messages vides, les () et $\langle\rangle$ sont omis), il se réduit par exemple de la manière suivante :

$$!a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{a} \mid \bar{b} \rightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{a} \mid \bar{c} \mid d \rightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{b} \mid \bar{b} \mid \bar{c} \mid \bar{c} \mid d$$

$$\rightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{c} \mid d \mid \bar{b} \mid \bar{c} \mid \bar{c} \mid d \rightarrow !a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid d) \mid \bar{c} \mid d \mid \bar{c} \mid d \mid \bar{c} \mid \bar{c} \mid d$$

On dispose alors d'un exemple de terme qui ne termine pas, sous la forme deux processus se répondant infiniment :

$$!a.\bar{b} \mid !b.\bar{a} \mid \bar{a} \quad \longrightarrow \quad !a.\bar{b} \mid !b.\bar{a} \mid \bar{b} \quad \longrightarrow \quad !a.\bar{b} \mid !b.\bar{a} \mid \bar{a}$$

1.2 Le premier système de types de [DS06]

L'objectif du typage est d'obtenir un ordre multi-ensemble sur les canaux qui va décroître strictement avec un pas de calcul. La terminaison sera ainsi garantie. L'idée de [DS06] est d'associer des niveaux aux canaux de telle manière qu'une réplication sur un canal ne contienne dans sa continuation que des canaux de niveau strictement inférieur.

Par exemple, le terme $!a.(\bar{b} \mid \bar{b} \mid \bar{c}) \mid !b.(\bar{c} \mid \bar{c}) \mid \bar{a} \mid \bar{b}$ admet pour type $a \mapsto 3$, $b \mapsto 2$, $c \mapsto 1$. Son exécution termine car son poids décroît avec les transitions : Le vecteur nombre d'émission de poids 3, nombre d'émission de poids 2, nombre d'émission de poids 1 suit l'évolution $[1,1,0] \rightarrow [0,3,1] \rightarrow^* [0,0,7] \not\rightarrow$.

Dans cette sous-partie, nous verrons en 1.2.1 comment inférer des niveaux adéquats pour les canaux.

Mais il faut se méfier car du fait des passages de messages, un canal peut désigner ou être désigné par deux noms différents. Connaître quel canal peut instancier quelle variable liée est nécessaire pour typer correctement un terme.

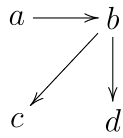
Le terme $p(x,y).!x.\bar{y} \mid \bar{p}\langle u,v\rangle \mid !v.\bar{u} \mid \bar{u}$ illustre ce danger. En oubliant que x peut être u et y v, on typerait un terme qui devient $!u.\bar{v} \mid !v.\bar{u} \mid \bar{u}$ qui ne termine pas.

Nous regarderons donc en 1.2.2 comment reconnaître les canaux jouant le même rôle.

1.2.1 Allocation de niveaux

Pour inférer les niveaux des canaux, on crée un *graphe de domination*. Ses sommets sont les canaux et une arête relie a à b si une émission sur b est réalisée à la suite d'une réplication sur a.

Par exemple, le terme $!a.\bar{b} \mid !b.\bar{c}.\bar{d} \mid c.\bar{d}$ a pour graphe de domination :



S'il existe un tri topologique sur ce graphe alors le terme est typable (et donc termine). Ce tri topologique induit une numérotation des canaux que l'on appelle leur *niveau*.

1.2.2 Reconnaissance de la structure des canaux

Sans passage de nom, c'est-à-dire si tous les canaux ne transportaient rien, associer un niveau à un canal sous les hypothèses de la partie 1.2.1 suffit à garantir qu'il s'arrêtera.

Quand il y a concurrence pour la réception d'un message comme par exemple dans $\overline{p}\langle a\rangle \mid \overline{p}\langle b\rangle \mid p(x)$ où a et b peuvent tous deux remplacer x, il faut considérer a, b et ce x comme potentiellement équivalents lors du typage et leur donner le même niveau.

Un parcours du processus afin de faire ce regroupement va donc devoir précéder l'allocation de niveaux.

Le regroupement des types doit se propager récursivement aux arguments potentiels des canaux impliqués. Par exemple, dans le terme $\bar{p}\langle a\rangle.\bar{p}\langle b\rangle \mid \bar{a}\langle u\rangle \mid \bar{b}\langle v\rangle$ dire que a et b sont identiques n'est pas suffisant car u et v doivent aussi être regroupés.

Nous profiterons de ce calcul pour vérifier simultanément que les processus pouvant être représentés par la même variable liée ont le même nombre d'arguments et pour vérifier le type des arguments afin d'éviter par exemple le terme $\bar{p}\langle a\rangle.\bar{p}\langle b\rangle \mid \bar{a}\langle\rangle \mid \bar{b}\langle v\rangle$ où p est susceptible d'émettre deux canaux de nature différente : a, qui est zéroadique (des messages vides sont émis dessus), et b, qui est monadique (des messages singleton sont émis dessus).

1.3 Limite de la démarche et solutions apportées

1.3.1 Un système de types trop contraignant

L'analyse exposée à la partie 1.2 est correcte, mais parfois pas assez subtile. Le terme

$$!p(a,b).a.(\overline{p}\langle a,b\rangle \mid \overline{b}) \mid \overline{p}\langle u,v\rangle \mid \overline{u} \qquad (1)$$

termine car il consomme un p et un a et libère un p et un b. Pourtant la présence d'une émission sur p sous la réplication sur p va faire échouer le typage. Cet exemple est la schématisation simplifiée de la structure de "simple table" dont on se couramment en π-calcul. Il faut donc chercher un moyen de ne pas le rejeter.

1.3.2 Evolution possible du contrôle statique de la terminaison

Concevoir des systèmes de types les plus permissifs possibles tout en conservant la preuve de la terminaison est l'objet de la thèse de Romain Demangeon, mon encadrant.

Une première idée (fonctionnant pour (1)) est de ne plus imposer que la réplication (ici p) soit de niveau strictement supérieur aux émissions (ici p et b) de sa continuation mais que le poids des émissions d'une réplication (toujours p et b dans l'exemple) soit supérieur aux poids des réceptions (p et a). Dans l'exemple on met un niveau à b inférieur à celui de a et le terme est typable. Ce typage est puissant mais son inference est NP-complète. (Preuve établie par Alain Darte et Romain Demangeon)

Une autre démarche exposée dans [DHS08] consiste à vérifier l'existence d'un bon ordre dynamiquement, c'est-à-dire au cours de l'exécution.

1.3.3 Vérification à l'exécution de la bonne réduction d'un terme annoté

De manière statique, on essaie de typer un processus dans le système [DS06]. En cas d'échec (comme dans le terme $!a.\overline{b} \mid !b.\overline{a}$ où les émissions sont nécessairement aux mêmes niveaux que les réplications parentes), au lieu de rejeter le terme au typage, il est décoré d'une assertion spécifiant la supposition faite. (L'exemple devient $!a.[a > b]\overline{b} \mid !b.[b > a]\overline{a}$.) Cette supposition est ensuite vérifiée à l'exécution pour conserver la terminaison du processus (ici puisque les deux assertions sont contradictoires, toute exécution mènera à une incohérence).

Il est nécessaire de modifier la sémantique du π-calcul pour nos processus annotés. Les même règles sont conservées mais chaque processus s'accompagne maintenant d'un ordre partiel sur les canaux représentant les suppositions faites antérieurement dans le calcul. Cet ordre est stocké sous forme d'un graphe. Une étape de calcul modifie en définitive un processus et son graphe en un processus et le graphe augmenté des nouvelles suppositions faites ou $\bot$ si une incohérence dans les assertions est trouvée.

Si $P \to P'$ alors $[\alpha < \beta]P, \mathcal{G} \to \begin{array}{l} P', \mathcal{G}' = \mathcal{G} \cup \{\{\alpha, \beta\}\} \text{ si } \mathcal{G}' \text{ est sans cycle} \\ \bot \text{ sinon} \end{array}$

Dans le cas de (1), le terme devient $!p(a,b).a.([p > p]\overline{p}\langle a,b\rangle \mid \overline{b}) \mid \overline{p}\langle a,b\rangle \mid \overline{a}$ qui est exécuté jusqu'à ce qu'on obtienne $!p(a,b).a.([p > p]\overline{p}\langle a,b\rangle \mid \overline{b}) \mid [p > p]\overline{p}\langle a,b\rangle \mid \overline{b}$ où la tentative d'insertion dans l'ordre de $p > p$ échoue.

[DHS08] donne la preuve que la version annotée d'un terme calculé grâce à la nouvelle sémantique n'a pas d'exécution divergente. L'arrêt sera causé par la fin du calcul ou son interruption car deux assertions contradictoires seront simultanément nécessaires.

Ce système d'annotations permet en réalité d'encoder le système [DS06] : On pourrait ne pas mettre de niveau aux canaux puis pour chaque émission écrire les assertions qu'elle implique pour qu'il n'y ai pas de divergence. Le système décrit ici est donc au moins aussi expressif que le système de 1.2. Néanmoins, cet encodage accroît inutilement la taille des graphes de niveaux (puisqu'il augmente le nombre d'assertions). L'exécution ralentirai sans accepter plus de termes terminants.

1.4 Les restrictions

Afin de clarifier la présentation, on a laissé de côté jusqu'ici un opérateur important du π-calcul, l'opérateur de *restriction*.

Jusque là, garantir l'unicité de l'usage d'un canal n'était pas possible. Or quand un serveur renvoie une réponse, il faut que ce soit le client qui a posé la question qui la re-

çoive et non un autre qui attend encore sa réponse. Cela est réalisé par l'intermédiaire de l'opérateur de restriction. Le client crée un canal "secret" qu'il donne au serveur pour qu'il lui réponde. Cette opération est notée (νc) si c est un nom restreint.

Les restrictions ajoutent des canaux frais au cours du calcul. Par conséquent, l'environnement change dynamiquement. Pour le typage [DS06], les canaux restreints sont considérés comme les autres canaux. Il n'en est pas de même pour le système de [DHS08] où une règle spécifique aux variables restreintes doit être rajoutée.

S'il existe un processus de la forme $!a.\dots(\nu c)\dots$ alors le niveau de a doit être strictement supérieur au niveau de c. Les placer au même niveau puis ajouter avant une émission sur c, $[a > c]$ compromet la correction de l'analyse de typage. Un processus tel que $!p(a,b).a.(\nu c)(\bar{p}\langle b,c\rangle \mid \bar{b}\langle\rangle) \mid \bar{p}\langle u,v\rangle \mid \bar{u}\langle\rangle$ mène à une divergence, en se rappelant récursivement sur un nom qu'il vient de créer. Ce terme ne peut être typé à l'aide de l'analyse que nous avons implémentée, mais les extensions de cette analyse auxquelles nous avons réfléchi durant le stage risqueraient de l'accepter, car elles introduiraient une annotation telle que le $[a > c]$ vu plus haut, ce qui conduirait à la construction d'une chaîne infinie. Une bonne prise en compte des restrictions dans l'analyse de type est donc nécessaire pour garantir la terminaison.

2 Implémentation

2.1 Fonctionnement d'un interpréteur

Mon implémentation s'appuie sur un interpréteur développé par Sébastien Briais dans le cadre de sa thèse. C'est un outil originalement adapté au spi-calcul qui est un modèle dérivé du π-calcul pour les communications sécurisées.

Cette partie détaille deux spécificités de cet interpréteur qui sont la clé d'une implémentation simplifiée et efficace d'un interpréteur de π-calcul.

2.1.1 Indices de De Bruijn

Les définitions originelles de modèles de langages de programmation, comme le λ-calcul ou le π-calcul, ne permettent pas de décrire les termes de façon canonique. Par exemple, les termes $a(x).c\langle x\rangle$ et $a(y).c\langle y\rangle$ désignent le même processus : celui qui renvoie sur le canal c un argument reçu sur le canal a. On considère ainsi les termes modulo *α-conversion*, c'est-à-dire renommage des variables liées.

De plus, la réduction rend parfois obligatoire un tel renommage. Ainsi, le terme $a(x).b(y).\overline{c}\langle x\rangle \mid \overline{a}\langle y\rangle$ doit, avant d'être réduit, se réécrire en $a(x).b(z).\overline{c}\langle x\rangle \mid \overline{a}\langle y\rangle$ pour que $b(z).\overline{c}\langle y\rangle$ soit correct (on obtiendrait sinon $b(y).\overline{c}\langle y\rangle$ alors que les y sont distincts).

De plus, réplication et restriction donnent lieu à la création de noms :

$$\begin{array}{rcl} !a.(\nu c)\overline{c} \mid \overline{a} \mid \overline{a} & \equiv & a.(\nu c_1)\overline{c}_1 \mid !a.(\nu c)\overline{c} \mid \overline{a} \mid \overline{a} \\ & \to & (\nu c_1)\overline{c}_1 \mid !a.(\nu c)\overline{c} \mid \overline{a} \\ & \equiv & (\nu c_1)\overline{c}_1 \mid a.(\nu c_2)\overline{c}_2 \mid !a.(\nu c)\overline{c} \mid \overline{a} \\ & \to & !a.(\nu c)\overline{c} \mid (\nu c_1)\overline{c}_1 \mid (\nu c_2)\overline{c}_2. \end{array} \quad (2)$$

Là encore, recourir à des noms frais et à l'α-conversion est nécessaire pour faire un calcul correct.

Afin de répondre à ces exigences, une solution connue sous le nom de *convention de Barendregt* est d'implémenter un générateur de noms et de réécrire les termes à chaque transition de telle manière que tous les noms liés aient des écritures distinctes. Cette étape est trop lourde, elle augmente le temps d'execution et fait exploser l'espace requis pour gérer les noms.

Un moyen plus efficace dû à N. De Bruijn est de désigner un nom par le nombre de lieurs (les restrictions et les arguments des réceptions) qu'il faut traverser pour atteindre sa definition : $a(x).(\nu y).\overline{x}\langle y\rangle$ se traduit en $0().\nu.\overline{1}\langle 0\rangle$ (on suppose que les variables libres proviennent de restrictions préfixes au terme que l'on omet, en toute rigueur il faudrait donc écrire $\nu 0().\nu.\overline{1}\langle 0\rangle$). La représentation d'un terme sous cette forme est unique et non équivoque.

La conservation des noms sous forme de chaîne de caractère pour la reconnaissance lors du parsing puis le stockage des types des termes sont faits par le maintien d'une liste des lieurs traversés jusqu'au point courant.

La réplication d'un terme en indice de De Bruijn ne nécessite pas de renommage. Par exemple, l'exécution de (2) est : $!0.\nu\bar{0} \mid \bar{0} \mid \bar{0} \rightarrow\rightarrow !0.\nu\bar{0} \mid \nu\bar{0} \mid \nu\bar{0}$. C'est pourquoi c'est la solution qu'implémente l'outil dont je suis parti.

Néanmoins, sous cette forme, le nom d'un canal dépend de sa position. La transformation de $\bar{a}\langle u\rangle.c(x).\bar{a}\langle x\rangle$ est $\bar{0}\langle 8\rangle.1().\bar{1}\langle 0\rangle$ (Pour rappel, le 8 pour désigner u représente que nous considérons que u est introduit par la neuvième restriction précédant le terme. Il est comme l'indice de toutes les variables libres arbitraires, seul importe qu'il soit unique et cohérent à tout moment.) On voit qu'ici a est représenté à la fois par 1 et 0 et que 1 représente à la fois c et a. Il faut donc des fonctions qui maintiennent la cohérence des termes lors de leur analyse et a fortiori de leur réécriture.

Par exemple, réaliser une communication supprime le lieur de la réception ce qui impose de diminuer tous les indices dans sa continuation tout en substituant les occurrences des variables liées par leurs valeurs : $4.P \mid \overline{4}\langle 12\rangle.Q \rightarrow low_all(P[12/0]) \mid Q$ (low_all diminue tout les indices non nul de 1.)

Nous reviendrons rapidement sur ces manipulations lors de la présentation de l'opération de *strip* (2.3.3).

2.1.2 Découpage générique du code

Partir de quelque chose de déjà implémenté qu'il fallait adapter fut grandement simplifié par la modularité du code à ma disposition. Hormis quelques modifications satellites, je n'ai eu qu'à coder l'algorithme d'inférence pour le typage et les nouvelles règles atomiques de la sémantique. L'analyse syntaxique et l'exécution du nouvel interpréteur s'en sont déduits de manière transparente.

Concrètement, il existe plusieurs définitions du π-calcul très légèrement différentes. Suivant que l'on autorise un seul ou plusieurs canaux à être transporté par un canal, ou encore que l'on parle de spi-calcul ou même de π-calcul symbolique, les règles de sémantique ne sont pas exactement les mêmes. Sébastien Briais a codé un parseur reconnaissant toutes ces variantes et un foncteur qui déduit des règles données en entrée l'interpréteur de ce calcul.

Il m'a donc suffit d'écrire les règles de la sémantique de [DHS08] et de modifier le parseur pour qu'il comprenne les assertions afin obtenir un interpréteur adéquat.

(Le typage et la bibliothèque de gestion d'ordre étaient quand même à réaliser par ailleurs.)

En réalité, le calcul où les messages sont des n-uplet de canaux (et non un canal) n'était pas implémenté. L'ajouter représenta mon premier travail.

2.2 L'inférence du système [DS06]

Comme nous l'avons vu en 1.2, l'implémentation du système de types se découpe en deux parties dont nous étudierons successivement la mise en oeuvre.

2.2.1 Concept de région pour représenter les canaux substituables

Lors de l'inférence des types en suivant les règles présentées en 1.2.2, deux niveaux de similarité différents doivent être bien distingués : avoir le même type et avoir le même type simple.

Deux canaux ont le même *type simple* quand ils transportent la même chose. Par exemple dans $\overline{a}\langle c\rangle \mid \overline{b}\langle d\rangle \mid \overline{c}\langle\rangle \mid \overline{d}\langle\rangle$, a et b sont tous les deux des canaux transportant un nom de canal ne transportant rien. Mais, à aucun moment, l'un et l'autre ne sont en concurrence pour instancier une réception. Les niveaux de deux canaux de même type simple sont indépendants.

Deux canaux sont de même type quand ils ont le même usage. Dans le terme $\overline{a}\langle c\rangle \mid \overline{a}\langle b\rangle \mid a(x).P$, b et c sont de même type car ils sont en concurrence pour remplacer x dans P. Cette notion implique d'avoir le même type simple mais deux canaux de même type doivent à tout moment avoir le même niveau.

Inférer le type va revenir pour nous à regrouper les noms de même usage dans une même *région* qui contiendra en outre une *étiquette* spécifiant le type simple de ces membres.

Matériellement, on appelle *étiquette* une valeur écrite en mémoire qui définit la structure d'un canal (Par exemple : "Canal ne transportant rien", "Canal transportant les canaux de type {pointeur vers cette région}"). Un type est donné par un pointeur vers une étiquette ; être du même type revient à pointer vers la même étiquette. Par souci d'efficacité, nous utilisons des éléments qui spécifient leur région plutôt que la structure naturelle de région donnant les éléments qu'elle contient. Cette solution n'est possible que parce que les régions constituent une partition des noms.

Les étapes pour réaliser ces regroupements et en vérifier la cohérence sont les suivantes. Initialement, on crée une étiquette de type générique pour chaque variable. Ensuite, on parcourt le terme. Pour chaque canal rencontré, on précise son étiquette et vérifie la compatibilité avec ses usages précédents. Si on infère que deux canaux ont le même type, alors on les fait pointer vers la même étiquette et on infère que leur arguments potentiels sont de même type (on s'appelle récursivement sur leurs régions d'arguments). A la fin, l'ensemble des variables qui pointent vers une même étiquette correspond à une classe d'équivalence pour la relation "avoir le même usage en tant que nom".

2.2.2 Allocation de niveau

Une fois ces régions constituées, il faut les hiérarchiser en suivant les principes exposés dans la partie 1.2 : une communication sur un canal donné ne doit libérer que des émissions des canaux de niveau inférieur.

On crée un graphe de domination de la manière détaillée en 1.2.1 : on met une arête entre la région d'une réplication et les régions de toutes les émissions de sa continuation.

On cherche ensuite les composantes fortement connexes de ce graphe : si chaque arête est seule dans sa composante, alors le terme est typable. On associe comme niveau son numéro dans l'ordre topologique associé (puisque l'algorithme de recherche de composante fortement connexe donne directement un ordre topologique s'il est appliqué à un graphe sans circuit).

2.3 Adaptation des termes et mise en oeuvre de la sémantique annotée décrite en 1.3.3

2.3.1 Sémantique modifiée et maintien d'ordres

On défini des règles de sémantique pour le π-calcul qui transforment un processus en un autre processus. Pour garantir l'arrêt, nous les avons étoffées et nous devons transformer un processus, des assertions et des informations de hiérarchie en un processus et des informations de hiérarchie mises à jour ou $\perp$ si une incohérence est trouvée.

Pour ce faire, il faut maintenir à chaque étape du calcul un graphe sans circuit de hiérarchisation des canaux.

Par ailleurs, nous voulons qu'il n'existe pas de chaîne de réduction infinie. Pour vérifier cela, il ne suffit pas de dérouler une exécution possible du calcul. Pour être sûr que nous ne pouvons pas trouver de chemin ne terminant pas, il nous faut exécuter toutes les transitions possibles à chaque étape puis regarder toutes les transitions possibles depuis chacune d'elles.

Pour ne pas faire exploser le coût du calcul, il nous fallait donc une implémentation fonctionnelle des graphes contenant des fonctions efficaces de maintien d'ordre à l'insertion d'arètes. Cette insertion d'une assertion dans un ordre en maintenant sa cohérence est tiré de [MNR96].

Notre module se base sur la bibliothèque *ocamlgraph* développée par JC Filliatre (Univ. Orsay).

2.3.2 Augmentation des possibilités du logiciel

Le code existant donnait l'ensemble des processus à un pas de calcul d'un terme fourni. Il fallait donc en déduire une fonction qui poursuive le calcul jusqu'à sa fin. Pour éviter l'explosion du nombre de termes, il a aussi fallu éliminer les redondances en utilisant une structure d'ensemble.

Il y a ici un gros travail autour de l'égalité de processus et de couple (processus, graphe de contrôle) qui ne fut pas l'objet d'étude durant ce stage.

2.3.3 Mutation des termes

Les variables liées nous posent problème pour réaliser notre graphe. Pourtant, une restriction qui n'est pas sous une réplication peut très bien (à α-conversion près) être considérée comme une variable libre de plus. Le terme $a(x).(\nu c)P$ se transforme en $(\nu c)a(x).P$ etc ...

Avant chaque étape de calcul, nous allons donc "sortir" les restrictions pour les intégrer à l'environnement des variables libres. Comme nous travaillons en indices de De Bruijn et que les lieurs sont déplacés, il faut que les indices changent convenablement. C'est la fonction *low_subst* qui code la transformation. Elle fait décroître les indices des variables libres du nombre de restrictions qu'il y avait et change les indices des variables liées en des numéros supérieurs au nombre de variables libres afin de représenter l'insertion des restrictions en fin de liste des variables libres.

2.4 Exemples

On reprend ici tous les exemples rencontrés au fur et à mesure de ce rapport pour illustrer comment réagit mon outil face à eux.

Les commandes sont les suivantes :

type tente de typer le terme donné et renvoie les niveaux des canaux. En cas d'échec, il rajoute les assertions adéquates.

calc type le terme puis réalise le calcul. Il renvoie tous les termes atteignables en une communication en séparant les ensembles par une ligne de tirets. Il marque qu'il y sa une incohérence dans les assertions si c'est la cause de son arrêt.

```
##Canal aux usages incompatibles
?> type 'a<a>
This term can't be typed

##Exemple de structure de canal sur un terme sans problème
?> type a(x)|'a<c>
a : #0(_a), c : _a

##Communication sur un canal
?> calc a(x)|'a<c>
Calculus:
(a(x).0 | 'a<c>.0)
-----
(0 | 0)
-----

##Exemple de terme typable avec réplication
?> type !a.'b|!b.'c|'a
a : #2(), b : #1(), c : #0()

##Calcul montrant le fonctionnement des réplications
?> calc !a.'b|!b.'c|'a
```

```
Calculus:
((! a.'b.0 | ! b.'c.0) | 'a.0)
-----
((('b.0 | ! a.'b.0) | ! b.'c.0) | 0)
-----
(((0 | ! a.'b.0) | ('c.0 | ! b.'c.0)) | 0)
-----

## Exemple du terme non terminant canonique
?> type !a.'b|!b.'a|'a
a : #1(), b : #1()

##Calcul et arrêt grace à la sémantique étendue
?> calc !a.'b|!b.'a|'a
Calculus:
((! a.[b < a] 'b.0 | ! b.[a < b] 'a.0) | 'a.0)
-----
((([b < a] 'b.0 | ! a.[b < a] 'b.0) | ! b.[a < b] 'a.0) | 0)
-----
(((0 | ! a.[b < a] 'b.0) | ([a < b] 'a.0 | ! b.[a < b] 'a.0)) | 0)
cycled_execution

#Importance des usages
##Cas sans problème
?> type p(a,b).!a.'b|'p<u,v>
p : #0(#2(), #1()), u : #2(), v : #1()

##Arrivée du problème
?> type p(a,b).!a.'b|'p<u,v>|!v.'u
p : #0(#2(), #2()), u : #2(), v : #2()

##Exemple de calcul où restriction+ réplication crée des noms
?> calc !a.(^c)'b<c>|b(x).b(y).!x.'y|'a
Calculus:
((! a.(^c)'b<c>.0 | b(x).b(y).! x.[y < x] 'y.0) | 'a.0)
-----
((((^c)'b<c>.0 | ! a.(^c)'b<c>.0) | b(x).b(y).! x.[y < x] 'y.0) | 0)
-----
(((0 | ! a.(^c)'b<c>.0) | b(y).! c1.[y < c1] 'y.0) | 0)
-----
```

Conclusion

Bilan

Mon vérificateur de type fonctionne mais il n'est certainement pas réutilisable par d'autres en l'état. En effet, le code fourni et produit n'est pas commenté et il n'existe pas de manuel d'utilisation. De plus, certains exemples ont une réponse étrange sûre-

ment due à un bug. Ce stage a néanmoins été très enrichissant pour moi. Il m'a permis de bien comprendre le π-calcul et plus généralement d'appréhender à la fois les modes de raisonnement pour prouver la terminaison et leur mise en oeuvre concrète. Enfin, j'ai découvert la difficulté que j'avais à rédiger et ai bénéficié de nombreux conseils et explications pour y remédier.

Pistes futures

Nous avons dit en 1.3.2 qu'il est trop réducteur de dominer les émissions par la première réception d'une réplication. Raisonner sur toutes les réceptions situées entre la réplication et la première émission qui la suit est parfois nécessaire pour typer un terme.

Pour une réplication $!P$ donnée, prenons R le multi-ensemble des réceptions de P précédant la première émission de P (si $P =!a_1(x_1).a_2(x_2)\dots a_k(x_k).\bar{b_1}\langle y_1\rangle.C$ alors $R = \{a_i\}_i$) et E le multi-ensemble des émissions de P. Imposer que tous les éléments de $R \setminus E$ dominent chacun des éléments de $E \setminus R$ est une approximation inférable en temps polynomial du cas précédant. Il est apparu durant le stage avec l'étude de (1) en particulier qu'il serait pertinent de l'implémenter.

D'autre part, la sémantique de [DHS08] modélise mal le calcul distribué car elle contient un ordre que tout les processus doivent connaître. Il faudrait donc rechercher comment mettre en oeuvre cette sémantique de manière réellement distribuée. Cela reviendrait à ce que les processus s'échangent leurs connaissances partielles du graphe lors de leurs communications et que cela suffise à vérifier la cohérence de l'ensemble. Déterminer quelles sont les informations que doit posséder un processus pour détecter les incohérence qu'il génère est la clé pour résoudre ce problème.

Références

[DHS08] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Static and dynamic typing for the termination of mobile processes. to appear in Proc. IFIP TCS'08, 2008.

[DS06] Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7) :1045–1082, 2006.

[MNR96] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59 :53–58, 1996.

Remerciement

Je remercie l'équipe PLUME pour l'environnement de stage détendu qu'elle m'a offert. Je remercie vivement mes encadrants pour leur disponibilité de tous les instants. Surtout, je remercie Daniel Hirschkoff et Aurelien Pardon pour leur infinie patience et leur considération lors de la rédaction du rapport sans laquelle je n'aurais jamais produit ce que vous avez lu.